

# Pepsi 2.0 Unpacking



Luca D'Amico

<https://www.lucadamico.dev>

21-Aug-2023

# Summary

<b>Abstract.....</b>	<b>3</b>
<b>Environment And Tools .....</b>	<b>4</b>
<b>Initial Analysis .....</b>	<b>5</b>
OSINT.....	5
High Detection Rate on Virus Total .....	7
Packer Detection .....	7
<b>Unpacking .....</b>	<b>9</b>
How Pepsi 2.0 Works? .....	9
Dumping: Method A - Manual Fix.....	16
Dumping: Method B - Automatic.....	18
Dumping: Method C - Dumping Unpacked File From Temporary Memory .....	19
<b>Conclusion .....</b>	<b>23</b>
<b>Credits .....</b>	<b>23</b>

## Abstract

This document is dedicated to the analysis and unpacking of Pepsi 2.0: a packer from the underground scene about which very little public information is available.

As described below, this packer has some limitations and various problems, but uses some interesting techniques.

The examined binary was extracted from the unpackme collection of tuts4you forum (<https://forum.tuts4you.com/files/file/1314-tuts-4-you-unpackme-collection-2016>), it is therefore available for free and legally.

## Environment And Tools

Pepsi analysis was performed on a virtual machine running Windows XP SP3. The following tools were used:

- PEiD, DIE & CFF Explorer: to obtain information about the binary.
- x32dbg & Scylla: debugging and unpacking.
- HxD: patching to the dump to fix the header applying the first proposed method.
- Lord PE: to fix the dump using the third proposed method.

The unpackme has the following SHA256:

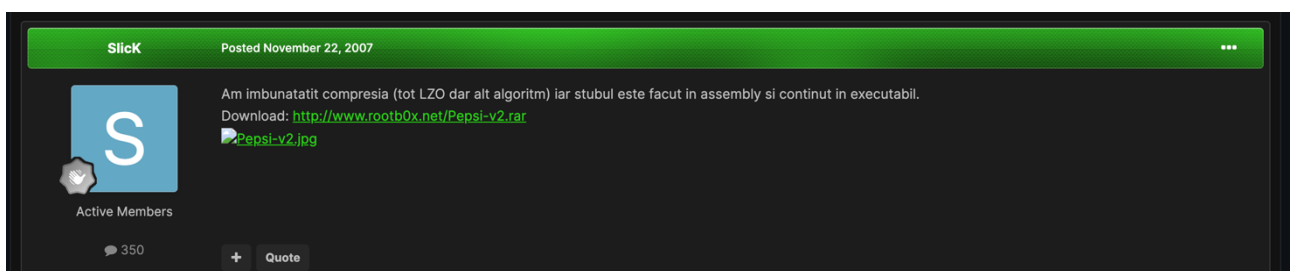
79288c042048afd61d3ddec9a75b8bedf1830adfc015c873d676ff4782d2a339

## Initial Analysis

In this section we will describe various analyses carried out on the binary to obtain useful information.

### OSINT

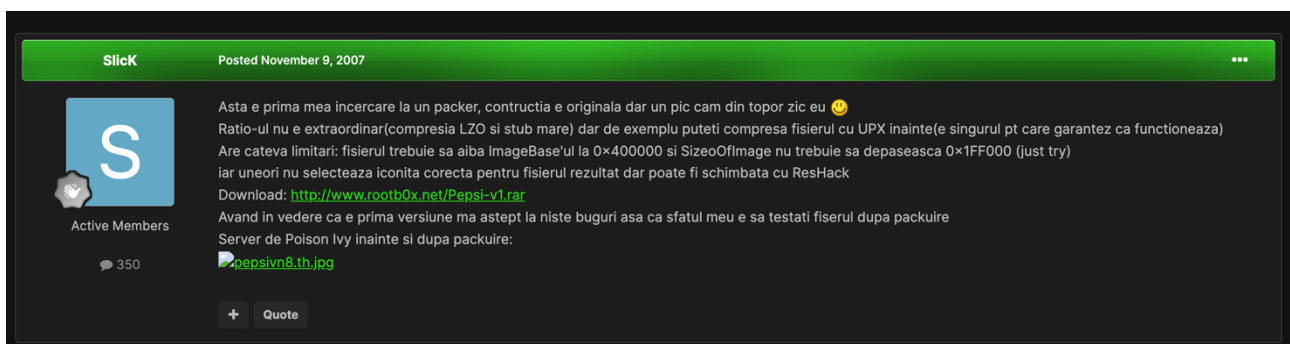
We don't have much information about this packer. By doing a targeted search on Google using the keywords “Pepsi” “Slick” and “packer”, the only significant reference comes from a Cyber Security forum in Romanian sent by the creator of Pepsi and dating back to November 22, 2007, with the title “Pepsi Packer v2”:



Translating the post with Google Translate we get:

*“I improved the compression (also LZO but another algorithm) and the stub is made in assembly and contained in the executable.”*

The provided link at the end of the post is unfortunately no longer accessible, but by doing a new search on Google limiting the results to the forum in question (using the keyword site:https://rstforums.com/), we can also find the related post that advertises the first version of the packer, titled “Pepsi Packer v1”:



Again, using Google Translate, the translation is as follows:

*“This is my first attempt at a packer, the construction is original, but a bit out of the ordinary, I’d say :)”*

*The ratio is not extraordinary (LZO compression and big stub) but for example you can first compress the file with UPX (it's the only one I guarantee that works)*

*It has some limitations: the file must have the ImageBase at 0x400000 and the SizeOfImage must not exceed 0x1FF000 (just try)*

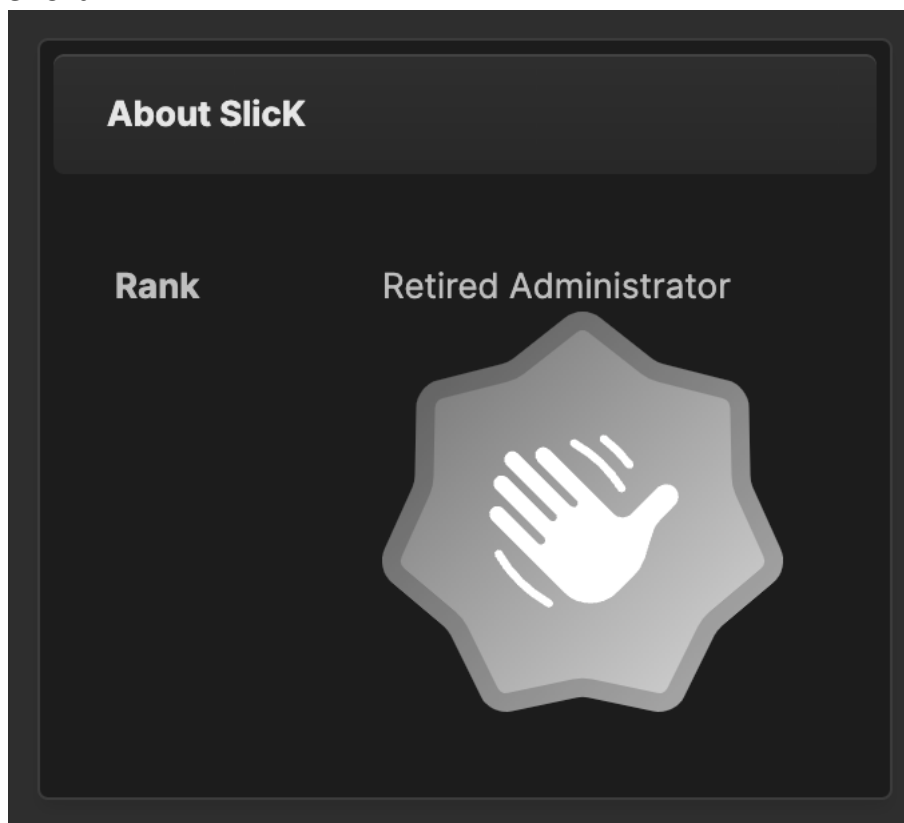
*and sometimes it does not select the correct icon for the resulting file, but it can be changed with ResHack*

*Download: ...*

*Considering that it is the first version, I expect some bugs, so my advice is to test the file after packaging”*

This document examines the V2 version of Pepsi, but I still thought it would be interesting to add this information about the V1 as these issues are still present.

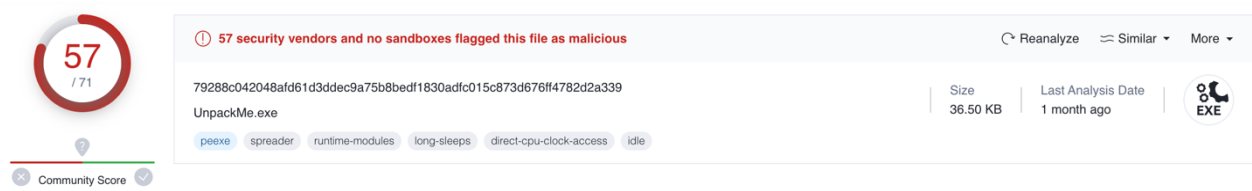
One last curiosity about SlicK, the author of Pepsi, can be obtained by looking at his profile on the forum:



Apparently, he is the former admin of this forum, and he was last logged in on November 12, 2011. I hope he is fine and that he will read this paper one day 😊.

## High Detection Rate on Virus Total

This packer has a very high detection rate on Virus Total:



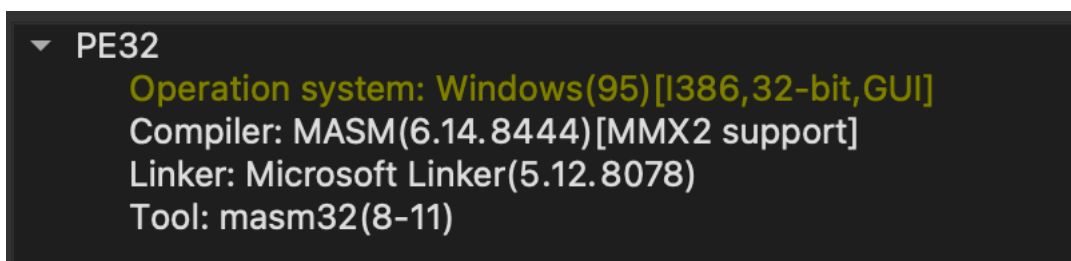
The screenshot shows the Virus Total interface for the file 'UnpackMe.exe'. A circular badge on the left indicates a detection rate of 57 out of 71 vendors. A red warning icon at the top states '57 security vendors and no sandboxes flagged this file as malicious'. The file's SHA-256 hash is 79288c042048afd61d3ddec9a75b8bedf1830adfc015c873d67ff4782d2a339. The file size is 36.50 KB and it was last analyzed 1 month ago. The file type is EXE. Behavioral tags include 'peexe', 'spreader', 'runtime-modules', 'long-sleeps', 'direct-cpu-clock-access', and 'idle'. A 'Community Score' of 1 is shown at the bottom left.

57 antivirus softwares detect it as malware. VT assigned the following name to the possible threat: trojan.baryx/backdoorx.

While it's possible that the techniques used by the packer could cause false positives, it's also likely that Pepsi was employed (not by its original author) over time to hide some malware.

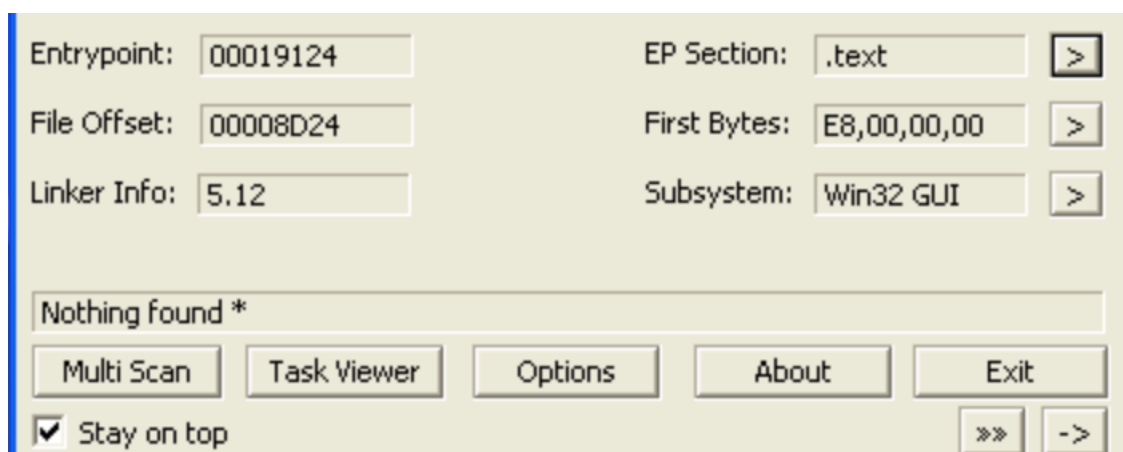
## Packer Detection

Pepsi is completely invisible to the most notorious Packer Detectors. Detect It Easy only detects that the binary was written using MASM:



The screenshot shows the PE32 analysis window of Detect It Easy. The text displayed is: 'PE32', 'Operation system: Windows(95)[I386,32-bit,GUI]', 'Compiler: MASM(6.14.8444)[MMX2 support]', 'Linker: Microsoft Linker(5.12.8078)', and 'Tool: masm32(8-11)'.

PEiD detects nothing:



The screenshot shows the PEiD interface. The fields are: Entrypoint: 00019124, EP Section: .text, File Offset: 00008D24, First Bytes: E8,00,00,00, Linker Info: 5.12, and Subsystem: Win32 GUI. A text box at the bottom contains the message 'Nothing found \*'. Below the text box are buttons for 'Multi Scan', 'Task Viewer', 'Options', 'About', and 'Exit'. At the bottom left, there is a checked checkbox for 'Stay on top' and navigation arrows at the bottom right.

A quick way to detect the presence of Pepsi is to verify that there is one section called “.pepsi”. In this screenshot this verification was done using CFF Explorer:

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.pepsi	00018000	00001000	00008891	00000200
.text	00001000	00019000	00000600	00008C00

Also, it seems that Pepsi is unable to keep the original icon of the programs it compresses. The final binaries have no icon until they are unpacked.



# Unpacking

In this section we will attack the binary, debugging and studying Pepsi's runtime behavior and finally extracting the original unpacked binary.

## How Pepsi 2.0 Works?

Let's load pepsi.exe in x32dbg and reach the EntryPoint:

00419124	E8 00000000	call pepsi.419129
00419129	6A 00	push 0
0041912B	E8 28040000	call <JMP.&GetModuleHandleA>
00419130	A3 04914100	mov dword ptr ds:[419104],eax
00419135	8B3D 04914100	mov edi,dword ptr ds:[419104]
0041913B	037F 3C	add edi,dword ptr ds:[edi+3C]
0041913E	81C7 F8000000	add edi,F8
00419144	8B57 08	mov edx,dword ptr ds:[edi+8]
00419147	8915 0C914100	mov dword ptr ds:[41910C],edx
0041914D	8B57 0C	mov edx,dword ptr ds:[edi+C]
00419150	0315 04914100	add edx,dword ptr ds:[419104]
00419156	8915 10914100	mov dword ptr ds:[419110],edx
0041915C	8B57 0C	mov edx,dword ptr ds:[edi+C]
0041915F	8915 08914100	mov dword ptr ds:[419108],edx
00419165	8B57 10	mov edx,dword ptr ds:[edi+10]
00419168	8915 14914100	mov dword ptr ds:[419114],edx
0041916E	6A 04	push 4
00419170	68 00100000	push 1000
00419175	FF35 0C914100	push dword ptr ds:[41910C]
0041917B	6A 00	push 0
0041917D	E8 EE030000	call <JMP.&VirtualAlloc>

As you can see, the current module handle is obtained by passing 0 (NULL) to the GetModuleHandleA API. The eax register will now contain the value 0x400000 (where the executable resides in memory). Various information from the header is then retrieved and saved. To make this part easier to understand, I assigned meaningful names to the various memory addresses and commented out the disassembly:

00419124	E8 00000000	call pepsi.419129	call \$0
00419129	6A 00	push 0	
0041912B	E8 28040000	call <JMP.&GetModuleHandleA>	
00419130	A3 04914100	mov dword ptr ds:[<imagebase>],eax	
00419135	8B3D 04914100	mov edi,dword ptr ds:[<imagebase>]	
0041913B	037F 3C	add edi,dword ptr ds:[edi+3C]	[edi+3c] = address of PE header
0041913E	81C7 F8000000	add edi,F8	start of first section header (.pepsi)
00419144	8B57 08	mov edx,dword ptr ds:[edi+8]	[edi+8] = virtual size of .pepsi
00419147	8915 0C914100	mov dword ptr ds:[<virtualsize_pepsi>],edx	
0041914D	8B57 0C	mov edx,dword ptr ds:[edi+C]	
00419150	0315 04914100	add edx,dword ptr ds:[<imagebase>]	[edi+c] = RVA of .pepsi
00419156	8915 10914100	mov dword ptr ds:[<virtualaddr_pepsi>],edx	RVA + imagebase = VA of .pepsi
0041915C	8B57 0C	mov edx,dword ptr ds:[edi+C]	
0041915F	8915 08914100	mov dword ptr ds:[<rva_pepsi>],edx	[edi+c] = RVA of .pepsi
00419165	8B57 10	mov edx,dword ptr ds:[edi+10]	
00419168	8915 14914100	mov dword ptr ds:[<rawsize_pepsi>],edx	[edi+10] = sizeofRawData
0041916E	6A 04	push 4	
00419170	68 00100000	push 1000	
00419175	FF35 0C914100	push dword ptr ds:[<virtualsize_pepsi>]	
0041917B	6A 00	push 0	
0041917D	E8 EE030000	call <JMP.&VirtualAlloc>	

At this point we find a call to VirtualAlloc, at the address 0x41917D, which will allocate a portion of memory as large as the virtual size of the ".pepsi" segment, in this case 0x18000.

We continue by analyzing the following portion of the disassembly:

```

0041917D E8 EE030000 call <JMP.&virtualAlloc>
00419182 0BC0 or eax,eax
00419184 75 07 jne pepsi.419180
00419186 6A 00 push 0
00419188 E8 C5030000 call <JMP.&ExitProcess>
0041918D A3 18914100 mov dword ptr ds:[419118],eax
00419192 6A 00 push 0
00419194 68 1C914100 push pepsi.41911C
00419199 FF35 18914100 push dword ptr ds:[419118]
0041919F FF35 14914100 push dword ptr ds:[419114]
004191A5 FF35 10914100 push dword ptr ds:[419110]
004191AB E8 C1010000 call pepsi.419371

```

If the memory allocation fails, the program will terminate by calling ExitProcess. Otherwise, the starting address of the newly allocated memory (obviously contained in the eax register) will be saved in [0x419118].

We also notice how this data is pushed onto the stack and used as a parameter in the call at the address 0x4191AB.

Before continuing, let's give meaningful names to the addresses to make the disassembly easier to understand:

```

0041917D E8 EE030000 call <JMP.&virtualAlloc>
00419182 0BC0 or eax,eax
00419184 75 07 jne pepsi.419180
00419186 6A 00 push 0
00419188 E8 C5030000 call <JMP.&ExitProcess>
0041918D A3 18914100 mov dword ptr ds:[<reserved_space_for_unpacking>],eax
00419192 6A 00 push 0
00419194 68 1C914100 push pepsi.41911C
00419199 FF35 18914100 push dword ptr ds:[<reserved_space_for_unpacking>]
0041919F FF35 14914100 push dword ptr ds:[<rawsize_pepsi>]
004191A5 FF35 10914100 push dword ptr ds:[<virtualaddr_pepsi>]
004191AB E8 C1010000 call pepsi.419371

```

By looking at the parameters passed to this call we can already imagine what will happen.

Indeed, we execute the function without entering it and then we have a look, in the memory dump window, the portion of memory previously reserved with the VirtualAlloc (which starts at the address saved in [0x419118], in this case 0x330000, which in turn we have renamed to "reserved\_space\_for\_unpacking"):

```

00330000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....yy..
00330010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....
00330020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00330030 00 00 00 00 00 00 00 00 00 00 00 00 B8 00 00 00 .....
00330040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..°..'!'.L!Th
00330050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
00330060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
00330070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode...$.
00330080 FA B9 CD 45 BE D8 A3 16 BE D8 A3 16 BE D8 A3 16 ú'IE%f.%f.%f.
00330090 30 C7 B0 16 86 D8 A3 16 42 F8 B1 16 BC D8 A3 16 0Ç°..0f.B0±.%0f.
003300A0 79 DE A5 16 BF D8 A3 16 52 69 63 68 BE D8 A3 16 yD%.ú0f.Rich%f.

```

This memory has been filled with an executable! We can already assume that this is the unpacked binary, but to be sure we need to continue our analysis.

Let's continue:

004191AB	E8 C1010000	call pepsi.419371
004191B0	0BC0	or eax,eax
004191B2	74 0C	je pepsi.4191C0
004191B4	83F8 F8	cmp eax,FFFFFFF8
004191B7	74 07	je pepsi.4191C0
004191B9	6A 00	push 0
004191BB	E8 92030000	call <JMP.&ExitProcess>
004191C0	FF35 18914100	push dword ptr ds:[<reserved_space_for_unpacking>]
004191C6	E8 C5000000	call pepsi.419290

We find another check which, in case of an error, forces the program to exit with the usual ExitProcess. Otherwise, if there are no problems, the memory address where the new executable resides is pushed onto the stack to be passed as a parameter to the function called at 0x4191C6.

This time we enter this call by clicking on Step Into.

We will arrive here:

00419290	55	push ebp
00419291	8BEC	mov ebp,esp
00419293	83C4 FC	add esp,FFFFFFFC
00419296	8B7D 08	mov edi,dword ptr ss:[ebp+8]
00419299	037F 3C	add edi,dword ptr ds:[edi+3C]
0041929C	8B57 28	mov edx,dword ptr ds:[edi+28]
0041929F	0315 04914100	add edx,dword ptr ds:[419104]
004192A5	8915 20914100	mov dword ptr ds:[419120],edx
004192AB	8B97 80000000	mov edx,dword ptr ds:[edi+80]
004192B1	0355 08	add edx,dword ptr ss:[ebp+8]
004192B4	8BFA	mov edi,edx
004192B6	E9 80000000	jmp pepsi.41933B
004192BB	BB 00000000	mov ebx,0
004192C0	035D 08	add ebx,dword ptr ss:[ebp+8]
004192C3	035F 0C	add ebx,dword ptr ds:[edi+C]
004192C6	53	push ebx
004192C7	E8 8C020000	call <JMP.&GetModuleHandleA>
004192CC	0BC0	or eax,eax
004192CE	75 0C	jne pepsi.4192DC
004192D0	53	push ebx
004192D1	E8 8E020000	call <JMP.&LoadLibraryA>
004192D6	0BC0	or eax,eax
004192D8	75 02	jne pepsi.4192DC
004192DA	EB 72	jmp pepsi.41934E

We know, as we have just said, that the address where the new executable (probably unpacked) resides had been passed as a parameter to this function, so it is located at [ebp+8]. The mov located at 0x419296 moves that address into the edi register. The instructions that follow first retrieve the PE header address ([edi+3C]) and immediately after that the RVA of the entry point ([edi+28]). In [419104] there is the address of the Pepsi imagebase (0x400000), and the add instruction (at 0x41929F) adds this address to the RVA of the entry point just recovered, to obtain its absolute address. This address is saved in [419120] and probably used in the future to perform the magic jump to pass control to the unpacked executable.

Don't be surprised if this doesn't currently make much sense since, as you may have noticed, the absolute address of the entry point we just computed is inside the .pepsi segment of the packer. The most logical explanation is that this memory area will soon be overwritten, probably with the data from the unpacked executable. We'll find out shortly.

Immediately afterwards, at 0x4192AB the RVA of the ImportDirectory ([edi+80]) is retrieved from the memory area where the unpacked executable resides and thanks to the add opcode, it is added to its imagebase to obtain its absolute address. This is necessary to build the IAT, in fact the following code does nothing but recover the libraries and functions necessary for the unpacked program to function correctly, using GetModuleHandleA, LoadLibraryA and GetProcAddress. Please note: the IAT is rebuilt inside the memory where the unpacked executable resides!

We figured out what this function does: calculate the absolute address of the entry point and reconstruct the IAT.

Continuing the analysis, we can see that also in this case there is a check to ensure that the just executed function has completed its task correctly. Immediately afterwards, at the address 0x4191E3, another call is made passing as parameters the address of the usual memory area where the unpacked executable resides (which now also has a valid IAT) and the packer imagebase, i.e. 0x400000 (stored at [419104]).

004191CB	83F8 01	cmp eax,1
004191CE	74 07	je pepsi.4191D7
004191D0	6A 00	push 0
004191D2	E8 7B030000	call <JMP.&ExitProcess>
004191D7	FF35 18914100	push dword ptr ds:[<reserved_space_for_unpacking>]
004191DD	FF35 04914100	push dword ptr ds:[419104]
004191E3	E8 63000000	call pepsi.41924B

Let's "Step Into" the function to analyze it:

0041924B	55	push ebp
0041924C	8BEC	mov ebp,esp
0041924E	83C4 FC	add esp,FFFFFFFC
00419251	8B75 08	mov esi,dword ptr ss:[ebp+8]
00419254	0376 3C	add esi,dword ptr ds:[esi+3C]
00419257	8B5D 0C	mov ebx,dword ptr ss:[ebp+C]
0041925A	035B 3C	add ebx,dword ptr ds:[ebx+3C]
0041925D	83BB 88000000 00	cmp dword ptr ds:[ebx+88],0
00419264	74 26	je pepsi.41928C
00419266	8D45 FC	lea eax,dword ptr ss:[ebp-4]
00419269	50	push eax
0041926A	6A 04	push 4
0041926C	6A 08	push 8
0041926E	56	push esi
0041926F	E8 08030000	call <JMP.&VirtualProtect>
00419274	8B93 8C000000	mov edx,dword ptr ds:[ebx+8C]
0041927A	8996 8C000000	mov dword ptr ds:[esi+8C],edx
00419280	8B93 88000000	mov edx,dword ptr ds:[ebx+88]
00419286	8996 88000000	mov dword ptr ds:[esi+88],edx
0041928C	C9	leave
0041928D	C2 0800	ret 8

From the parameters passed to the function, we know that the memory area where the unpacked executable resides starts at the address contained in [ebp+C], while in [ebp+8] we have the imagebase of the Pepsi packer we are analyzing. Therefore, before calling VirtualProtect, we will have the value 0x400080 in the esi register (address of the PE header of the packer) and in ebx the value 0x3300B8 (address of the PE header of the unpacked binary).

The VirtualProtect therefore has the task of changing the memory access protection starting from address 0x400080 which thanks to the parameter 0x4 (PAGE\_READWRITE) will now be modifiable.

The edx registry is loaded with the size of the Resource Directory of the unpacked executable ([ebx+8C]). This value is written to the Pepsi header ([esi+8C]). The edx register is now loaded with the VA of the Resource Directory, still from the unpacked executable ([ebx+88]) and also in this case this value is overwritten to the one present in the Pepsi header ([edi+88]).

So, this function just copies the Resource Directory details (size and virtual address) from the header of the unpacked binary, overwriting those present in the header of the packer with them.

Once we exit this call, we will immediately find another one:

004191E8	8B15 08914100	mov edx,dword ptr ds:[<rva_pepsi>]
004191EE	0115 18914100	add dword ptr ds:[<reserved_space_for_unpacking>],edx
004191F4	2915 1C914100	sub dword ptr ds:[41911C],edx
004191FA	FF35 1C914100	push dword ptr ds:[41911C]
00419200	FF35 18914100	push dword ptr ds:[<reserved_space_for_unpacking>]
00419206	FF35 10914100	push dword ptr ds:[<virtualaddr_pepsi>]
0041920C	E8 46010000	call pepsi.419357

Thanks to the meaningful names that we have given to the memory addresses, the parameters that are passed to this function is easy now to understand.

The RVA of the .pepsi segment (0x1000) is added to the start address of the memory where the unpacked executable resides. Immediately afterwards it is also subtracted from the size of its own segment (stored at [41911C]): so now at [41911C] there will be the value 0x17000, which is the first to be pushed into the stack.

The second value to be pushed is the starting address of the memory where the unpacked executable resides with the previous addition of 0x1000: probably the author of the packer is trying to reference the first segment after the header of the unpacked executable!

The third value pushed is the VA of the .pepsi segment.

With these details, we can already hypothesize what will happen as soon as we enter this call: the .pepsi segment of the packer will be overwritten with the data of the unpacked executable starting from 0x1000.

Let's find out if we're right by entering the function:

00419357	55	push ebp
00419358	8BEC	mov ebp,esp
0041935A	56	push esi
0041935B	57	push edi
0041935C	FC	cld
0041935D	8B75 0C	mov esi,dword ptr ss:[ebp+C]
00419360	8B7D 08	mov edi,dword ptr ss:[ebp+8]
00419363	8B4D 10	mov ecx,dword ptr ss:[ebp+10]
00419366	D1E9	shr ecx,1
00419368	F366:A5	rep movsw
0041936B	5F	pop edi
0041936C	5E	pop esi
0041936D	C9	leave
0041936E	C2 0C00	ret C

PERFECT! We are right! The cld opcode sets the forward direction of the copy. In the esi register we have the address 0x331000, which is the first segment of the unpacked executable. The value 0x401000, that is the VA of the .pepsi segment of the packer, is moved to the edi register. In ecx we have 0x17000, which is the size of the data to copy. The "rep movsw" opcode starts the copy.

Once out of this function we can say that we have clearer ideas: the unpacked executable with the rebuilt IAT has been copied to the .pepsi segment and the packer header has been suitably modified to have the correct Resource Directory data in relation to the original executable!

Once out of this function, there is very little left to analyze:

00419211	2915 18914100	sub dword ptr ds:[<reserved_space_for_unpacking>],edx
00419217	FF35 0C914100	push dword ptr ds:[<virtualsize_pepsi>]
0041921D	FF35 18914100	push dword ptr ds:[<reserved_space_for_unpacking>]
00419223	E8 42030000	call <JMP.&RtlZeroMemory>
00419228	68 00400000	push 4000
0041922D	FF35 0C914100	push dword ptr ds:[<virtualsize_pepsi>]
00419233	FF35 18914100	push dword ptr ds:[<reserved_space_for_unpacking>]
00419239	E8 38030000	call <JMP.&VirtualFree>
0041923E	FF15 20914100	call dword ptr ds:[419120]
00419244	6A 00	push 0
00419246	E8 07030000	call <JMP.&ExitProcess>

We have two calls, respectively to RtlZeroMemory and VirtualFree which do nothing but remove whatever trace of the unpacked executable from memory (smart huh?! 😊), followed by a call to [419120].

We already know that at [419120] there is the absolute address of the entry point calculated previously: this call does nothing but pass the control to the unpacked executable (that now lives in the .pepsi segment)!

In fact, once we enter this call we are here:

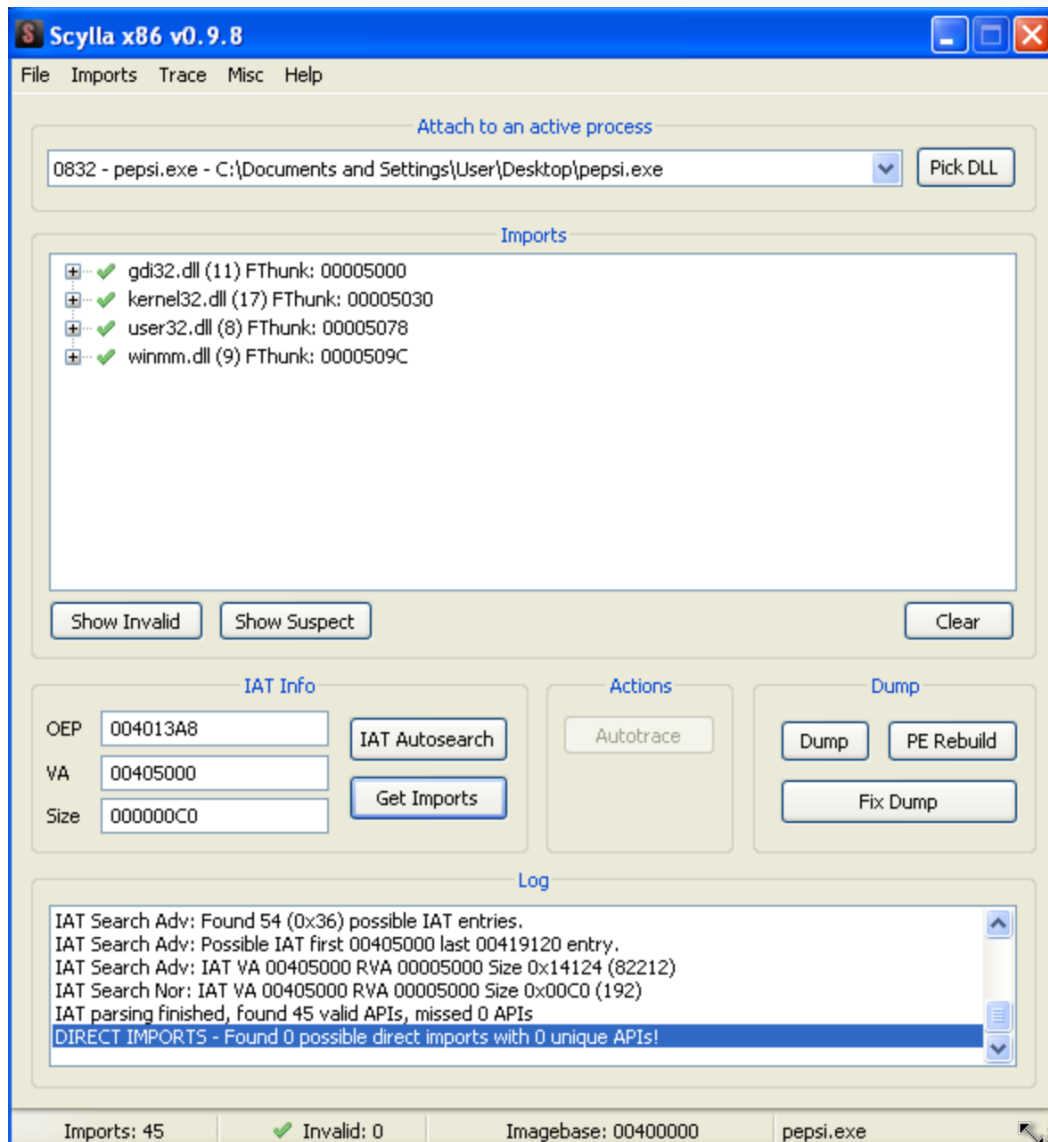
004013A8	56	push esi
004013A9	6A 0A	push A
004013AB	68 F4010000	push 1F4
004013B0	FF35 6D634000	push dword ptr ds:[40636D]
004013B6	E8 97000000	call <JMP.&FindResourceA>
004013BB	50	push eax
004013BC	50	push eax
004013BD	FF35 6D634000	push dword ptr ds:[40636D]
004013C3	E8 B4000000	call <JMP.&SizeofResource>
004013C8	A3 71634000	mov dword ptr ds:[406371],eax
004013CD	58	pop eax
004013CE	50	push eax

We have arrived at the OEP (original entry point)!! Now we can dump with Scylla.

Since there will be difficulties, related to the configuration of Scylla, I decided to write three separate sub-chapters about dumping, to illustrate all three possibilities.

## Dumping: Method A - Manual Fix

Once arrived at the OEP, let's open Scylla (just click the button with an 'S' icon from the x32dbg toolbar), enter the address of the OEP and click on "IAT Autosearch". We choose "no" when we are asked to use the advanced search method. We end up with this configuration:



Perfect, let's click on "Dump" and then on "Fix Dump".

Let's try to run our unpacked program and.... IT DOES NOT WORK!!

Why is it not working? The answer is quite simple: in the standard configuration of Scylla, the dump header is copied from the binary present on the disk and not from the one in memory. If you recall the Pepsi header was patched when the Resource



Directory size and VA values were changed! Our dump therefore does not have these updated values and we must proceed by changing them manually.

The fastest way is to open our dump (the one with the IAT fixed of course) in a hex editor like HxD and go to the offset where the data relating to the size and VA of the resource directory reside.

In our case we will have the VA of the Resource Directory at 0x109 and its size at 0x10C.

To obtain the correct values, restart the debugger and set a breakpoint where the modification of these data takes place, i.e.:

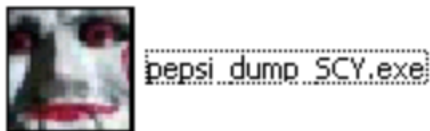
- 1) 0x41927A to get the size of the Resource Directory
- 2) 0x419286 to get the VA of the Resource Directory

So, we get 9D9C as size and E000 as VA.

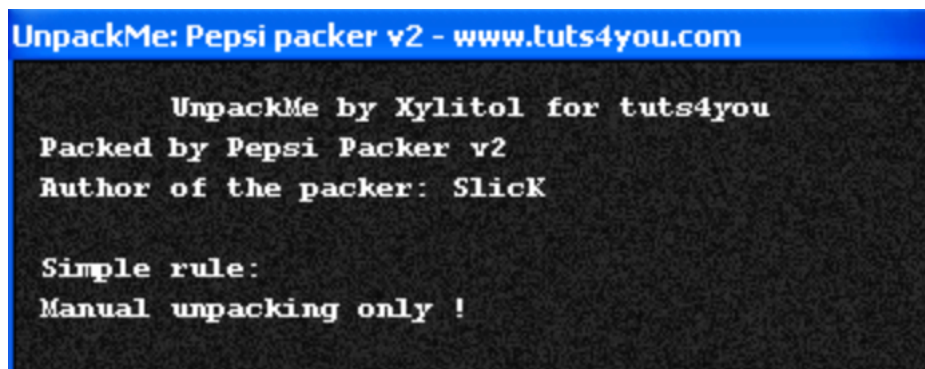
Let's proceed by patching the binary:

```
000000C0 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 .....
000000D0 00 B0 01 00 00 02 00 00 00 00 00 02 00 00 00 .°.
000000E0 00 00 10 00 00 10 00 00 00 10 00 00 10 00 00 .....
000000F0 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 .....
00000100 C4 A0 01 00 64 00 00 00 00 E0 00 00 9C 9D 00 00 ä . . d . . . à . . ce . 0
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

Let's save our patch and notice that now the executable also has an icon:



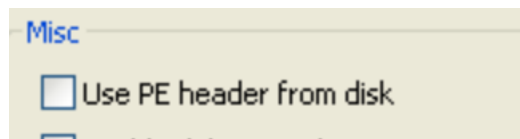
Let's run it and everything works! The packer has been removed:



## Dumping: Method B - Automatic

By the first method we manually patched the dump header to fix the Resource Directory values, now let's see how to properly configure Scylla to have a perfect dump without having to change anything.

Let's go back to the OEP and open Scylla, configure it exactly as we did previously, but before proceeding with the Dump, click on Misc and then on Options. From the window that will open, remove the check mark from "Use PE header from disk":



Continue by dumping regularly and clicking on "Fix Dump" to get a 100% working unpacked executable!

## Dumping: Method C - Dumping Unpacked File From Temporary Memory

This method is a bit more complicated than the ones discussed above, but I wanted to add it as I find it interesting.

During the analysis we discovered that for almost the entire execution of the packer the unpacked file is present in a temporary memory area and that after being copied to the .pepsi segment, it is deleted.

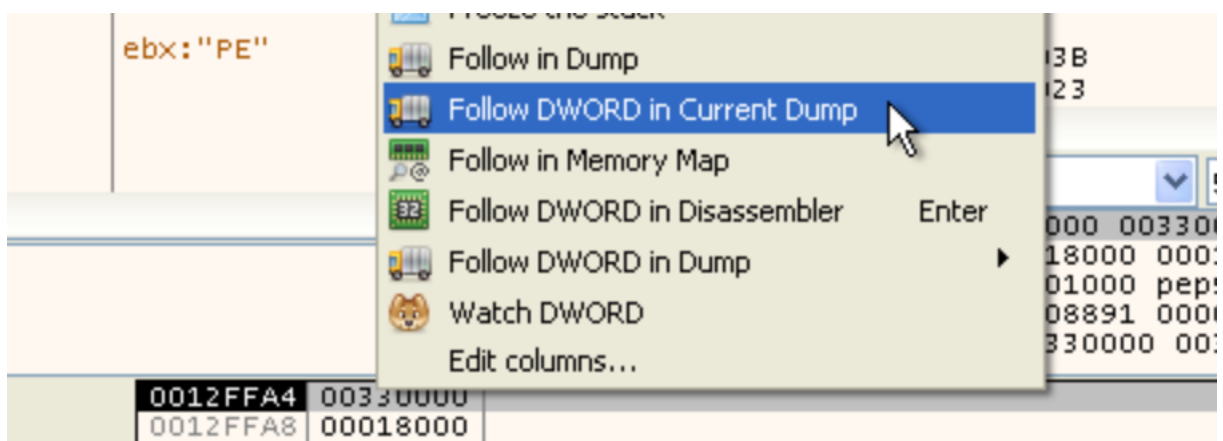
Also, we discovered that a function adds a valid IAT to the unpacked file located in this memory area.

We can dump this memory area by setting a breakpoint from the debugger just before it is overwritten, i.e., where the call to RtlZeroMemory takes place:

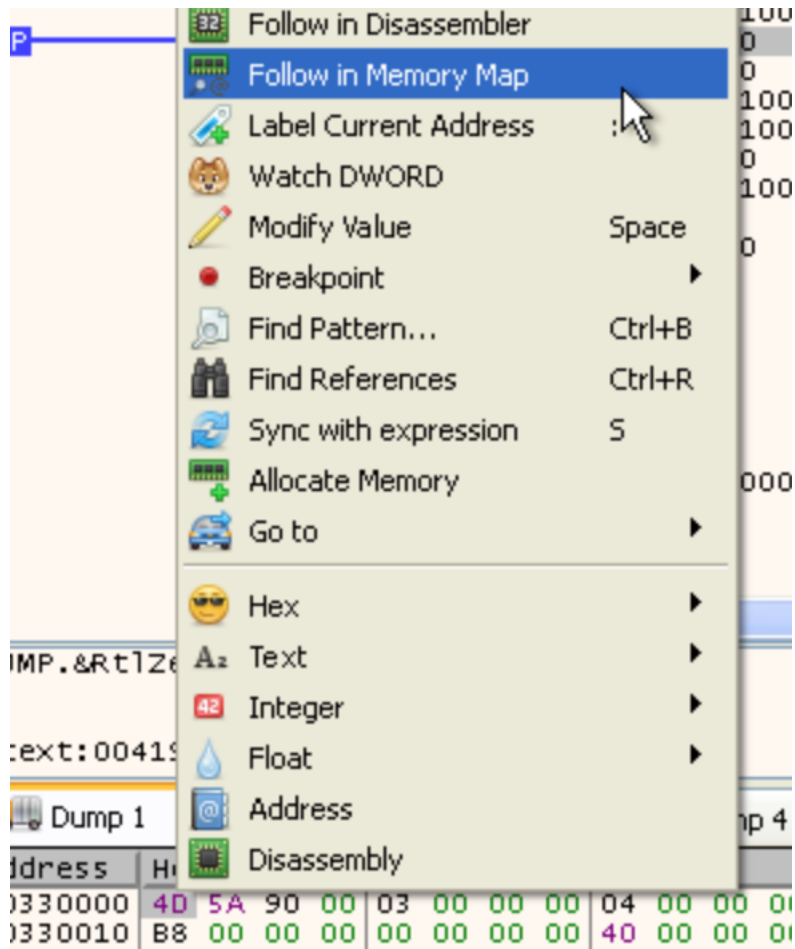
00419217	FF35 0C914100	push dword ptr ds:[<virtualsi
0041921D	FF35 18914100	push dword ptr ds:[<reserved_
00419223	E8 42030000	call <JMP.&RtlZeroMemory>
00419228	68 00400000	push 4000
0041922D	FF35 0C914100	push dword ptr ds:[<virtualsi

We are stopped just before the cleanup of this memory area.

Knowing that the address in question was the last one to be pushed, we can reach it in the Memory Map by right-clicking it on the related stack view window and choosing "Follow DWORD in Current Dump":



And immediately afterwards from the hex view, let's right-click and chose "Follow In Memory Map":



We will find ourselves on the Memory Map tab with the relevant memory section that interests us already selected. We can proceed by right-clicking and choosing “Dump Memory to File”:



We now have the unpacked file pulled from this temporary memory section, which also has a valid IAT, but there's a catch: the executable still can't run because it's in a memory-mapped state. We can check by opening it with CFF:

pepsi\_00330000.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...	Lin
000001B0	000001B8	000001BC	000001C0	000001C4	000001C8	000001CC	000001D0	00
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Wc
.text	000032A2	00001000	00003400	00000400	00000000	00000000	0000	00
.rdata	000004E6	00005000	00000600	00003800	00000000	00000000	0000	00
.data	00007384	00006000	00000200	00003E00	00000000	00000000	0000	00
.rsrc	00009D9C	0000E000	00009E00	00004000	00000000	00000000	0000	00

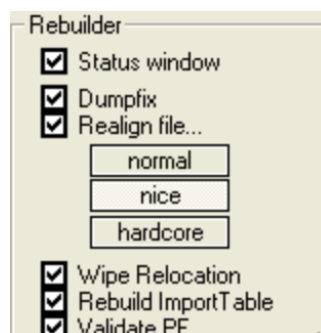
This section contains:  
Code Entry Point: 000013A8

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000B80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000B90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000BA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000BB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000BC0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000BD0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000BE0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000BF0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000C00	55	8B	EC	83	C4	B0	81	7D	0C	10	01	00	00	75	3F	C7	U i A* }    ..u?Ç
00000C10	05	1D	61	40	00	7C	00	00	00	C7	05	25	61	40	00	00	a@. ...Ç %a@..
00000C20	00	00	00	C7	05	21	61	40	00	02	00	00	00	FF	75	08	...Ç la@. ...ÿu
00000C30	E8	71	04	00	00	50	E8	0B	01	00	00	6A	00	6A	2E	6A	èq ..Pè   ..j.j.j
00000C40	01	FF	75	08	E8	69	04	00	00	E9	F1	00	00	00	83	7D	vu èi ..éñ... }

As we can see all the sections are misaligned.

To fix this, we need to realign all the sections so that the binary can be loaded correctly by the Windows executable loader. This is done automatically by Scylla and is called "unmapping the dump". We can proceed manually or with help of a tool called Lord PE.

Let's launch Lord PE, click on options, and put the check mark on the "Dumpfix" item:




Now click on “Rebuild PE” and choose our dumped binary. We confirm by clicking on OK and we will have our executable realigned and fully functional!

We can verify that the various sections are now aligned correctly using CFF:

pepsi\_00330000.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...
00000138	00000140	00000144	00000148	0000014C	00000150	00000154	00000158
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word
.text	00004000	00001000	000032A2	00000200	00000000	00000000	0000
.rdata	00001000	00005000	000004E6	00003600	00000000	00000000	0000
.data	00008000	00006000	000001BE	00003C00	00000000	00000000	0000
.rsrc	00009D9C	0000E000	00009D9C	00003E00	00000000	00000000	0000

This section contains:  
Code Entry Point: 000013A8



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	55	8B	EC	83	C4	B0	81	7D	0C	10	01	00	00	75	3F	C7	U i Ä* }    ..u?Ç
00000010	05	1D	61	40	00	7C	00	00	00	C7	05	25	61	40	00	00	a@. ...Ç %a@..
00000020	00	00	00	C7	05	21	61	40	00	02	00	00	00	FF	75	08	...Ç !a@. ...ÿu
00000030	E8	71	04	00	00	50	E8	0B	01	00	00	6A	00	6A	2E	6A	èq ...Pè  ...j.j.j

Everything is aligned correctly.

## Conclusion

We have completed the analysis of this packer and documented three different methods of unpacking.

In summary, Pepsi does the following:

- 1) An area of memory is allocated for temporary use and the unpacked executable is extracted inside it (without the IAT, at the moment)
- 2) With the information contained in this memory area, the address of the OEP is calculated and temporarily saved. The IAT is also created and written to this memory area.
- 3) The Resource Directory information (VA and size) are extracted from the header of the unpacked executable and written to the Pepsi header.
- 4) The unpacked executable (starting from 0x1000) is copied to the .pepsi segment.
- 5) The memory area where the unpacked executable resides is cleaned and freed.
- 6) The OEP is called, and the execution of the original program begins.

Considering this is Slick's first attempt at creating a packer, I can only congratulate him 😊

## Credits

As always, I would like to thank the authors of the tools used in the document. Special thanks go to Slick for creating Pepsi and Xylitol for creating this unpackme.

Thanks also goes to you who read this paper! I hope you have enjoyed the analysis :)

If you have enjoyed this document and want to read more about unpacking, malware analysis and reverse engineering, visit my site:

<https://www.lucadamico.dev/>

Luca